Doc number: **P0126R2**
Revises: **P0126R1, N4195**
Date: 2016-03-13
Project: Programming Language C++, Concurrency Working Group
Reply-to: Olivier Giroux ogiroux@nvidia.com, Torvald Riegel triegel@redhat.com
Thanks-to: Jeffrey Yasskin

std::synchronic<T>

Atomic objects make it easy to implement inefficient synchronization in C++. The first problem that users typically have, is poor system performance under oversubscription and/or contention. The second is high energy consumption under contention, regardless of oversubscription.

At issue is a trade-off centered on resource arbitration for synchronization, placing in tension:

- The focus of modern platform architecture is on lowering total energy use.
- The focus of performance-critical software is on minimizing latency.

Implementations could do significantly better with more semantic information. There exists different native support for efficient polling on all major software and hardware platforms. We now propose “synchronic” objects, an atomic library abstraction for this diverse support.

For more background, see P0126R0 and Futexes are Tricky.

A simplifying abstraction

Synchronic objects make it easier to implement scalable and efficient synchronization using atomic objects. The easiest way to use a synchronic object is to declare an expected atomic value for synchronization, and notify when an atomic object should be compared against this value.

For example:

```
//similar to std::latch (n4538)
//using std::hardware_false_sharing_size (n4523)
class example {
  ...
  void sync_up_my_team() {
    if(count.fetch_add(-1)!=1)
      while(!released.load());
      sync.wait(released, true);
    else
      released.store(true);
      sync.notify_all(released, true);
  }
  ...
  alignas(hardware_false_sharing_size) atomic<int> count;
  alignas(hardware_false_sharing_size) atomic<bool> released;
  synchronic<bool> sync;
};
```

C++ Proposed Wording

Apply these edits to the working draft of the Concurrency TS on 03/03/2016, N4399.

Feature test macros

The `__cpp_lib_synchronic` feature test macro should be added.

29.2 Header <atomic> synopsis:

```
namespace std {
  namespace experimental {
  inline namespace concurrency_v2 {

    // 29.9, synchronic operations
    enum class wait_hint {
      optimize_latency,
      optimize_utilization
    };

    template <class T> class synchronic;

  } // namespace concurrency_v2
  } // namespace experimental
} // namespace std
```

29.9 Synchronic objects [atomics.synchronic]

1 Synchronic objects provide low-level blocking primitives used to implement synchronization with atomic objects. Class `synchronic<T>` encapsulates an efficient algorithm to wait until a condition is met, evaluated over a single object of the corresponding class `atomic<T>`. This facility neither requires nor provides mutual-exclusion between threads on its own.

2 The `notify_one` and `notify_all` member functions are notifying functions. The `wait`, `wait_for_change`, `wait_for_change_until` and `wait_for_change_for` member functions are waiting functions.

3 Concurrent executions of the notifying and waiting functions do not introduce data races. If they invoke a user-provided function, that function may still introduce data races.

4 Executions of waiting functions may block until they are unblocked by a notifying function, according to each function's effects.

5 [*Example:* A simple latch pattern:

```
atomic<int> count = 2;               // number of threads participating
atomic<bool> ready = false;
synchronic<bool> sync;

void sync_up_my_team() {             // invoked once each thread
  if(count.fetch_add(-1)!=1)
    sync.wait(ready, true);          // all but the last thread blocks
  else
    sync.notify_all(ready, true);    // last thread unblocks all the others
}
```

– *End Example.*]

6 [*Note:* Programs using synchronic objects may be susceptible to transient values, an issue known as the ABA problem, resulting in continued blocking if the wait function's condition is only temporarily met. – *End Note.*]

29.9.1 Class `synchronic` [atomics.synchronic.class]

```
namespace std {
  namespace experimental {
  inline namespace concurrency_v2 {

    template <class T>
    class synchronic {
    public:

      synchronic();
      ~synchronic();
      synchronic(const synchronic&) = delete;
      synchronic& operator=(const synchronic&) = delete;
      synchronic(synchronic&&) = delete;
      synchronic& operator=(synchronic&&) = delete;

      void wait(const atomic<T>& object, T desired,
        memory_order order = memory_order_seq_cst,
        wait_hint hint = wait_hint::optimize_latency) const;
      void wait_for_change(const atomic<T>& object, T current,
        memory_order order = memory_order_seq_cst,
        wait_hint hint = wait_hint::optimize_latency) const;

      template <class Clock, class Duration>
      bool wait_for_change_until(const atomic<T>& object, T current,
        chrono::time_point<Clock,Duration> const& abs_time,
        memory_order order = memory_order_seq_cst,
        wait_hint hint = wait_hint::optimize_latency) const;
      template <class Rep, class Period>
      bool wait_for_change_for(const atomic<T>& object, T current,
        chrono::duration<Rep, Period> const& rel_time,
        memory_order order = memory_order_seq_cst,
        wait_hint hint = wait_hint::optimize_latency) const;

      void notify_all(atomic<T>& object, T value,
        memory_order order = memory_order_seq_cst);
      template <class F> void notify_all(atomic<T>& object, F func);

      void notify_one(atomic<T>& object, T value,
        memory_order order = memory_order_seq_cst);
      template <class F> void notify_one(atomic<T>& object, F func);
    };

  } // namespace concurrency_v2
  } // namespace experimental
} // namespace std
```

1 [*Note:* `synchronic<T>` probably is not an empty type. – *End Note.*]

```
synchronic();
```

2 *Effects:* Constructs an object of type `synchronic<T>`.

3 *Throws:* `system_error` (19.5.6).

```
~synchronic();
```

4 *Requires:* No threads are blocked on `this`.

5 *Effects:*

1. May block until executions of notifying functions on `this` have completed. [*Note*: This property generally allows a program to destroy a synchronic object immediately after it synchronizes with it. To ensure correctness, the program should also generally avoid causing side-effects to the atomic object outside of notifying functions. – *end note.*][*Example:*

```
struct broken_one_time_channel {
    atomic<bool> set = false;
    synchronic<bool> sync;
    void receive() {
      sync.wait(set, true); // May see set == true before notify_all executes.
                            // Reads the message.
      delete this; // May occur before or during call to notify_all; lifetime of sync ends.
    }
    void send() {
                                                    // Writes the message
      set = true; // Uses set outside of notify_all; may allow sync to be destroyed before
next line.
      sync.notify_all(set, [](auto&){}); // error: races with end of sync lifetime.

      // OK: sync.notify_all(set,true);
      // OK: sync.notify_all(set,[](auto& a){ a = true;});
    }
  };
```

– *End Example.*]

2. Destroys the object.

```
void wait(const atomic<T>& object, T desired, memory_order order,
  wait_hint hint) const;
template <class F>
void wait_for_change(const atomic<T>& object, T current,
  memory_order order, wait_hint hint) const;
template <class Clock, class Duration>
bool wait_for_change_until(const atomic<T>& object, T current,
  chrono::time_point<Clock,Duration> const& abs_time, memory_order order,
  wait_hint hint) const;
```

6 Let the *return condition* be:

- for the first form of the function, `object.load(order) == desired;`
- for the second form, `object.load(order) != current;`

- for the third form, `object.load(order) != current`, or when the absolute timeout expires.

7 *Effects:* Each execution of a waiting function is performed as:

1. Invokes `object.load(memory_order_relaxed)`. [*Note:* This observes the result of an operation in the modification order of `object`. – *End Node.*]
2. Evaluates the *return condition* then, if it is satisfied, returns. The third form may return spuriously.
3. Blocks.
4. Is unblocked:
   - As a result of some notifying operations, as described in that function's effects.
   - In the third form, when the absolute timeout expires.
   - At the implementation's discretion.
5. Each time the execution unblocks, these effects repeat from step 1.

8 *Returns:* for the third form, `true` if the *return condition* was satisfied.

9 *Throws:* `system_error` (19.5.6).

10 *Remarks:* the value of `hint` is only informative. The value `optimize_latency` indicates to the implementation that latency overheads should be minimized for this operation, even if it increases resource utilization, and `optimize_utilization` indicates otherwise.

```
template <class Rep, class Period>
bool wait_for_change_for(const atomic<T>& object, T current,
  chrono::duration<Rep, Period> const& rel_time, memory_order order,
  wait_hint hint) const;
```

11 *Effects:* Equivalent to:

```
wait_for_change_until(object, current,
  chrono::steady_clock::now() + rel_time, hint);
```

```
void notify_all(atomic<T>& object, T value, memory_order order);
template <class F> void notify_all(atomic<T>& object, F func);
void notify_one(atomic<T>& object, T value, memory_order order);
template <class F> void notify_one(atomic<T>& object, F func);
```

12 *Requires:* `func` is callable with the signature `void(atomic<T>&)` and does not invoke a member function on `this`.

13 *Effects:*

1. Invokes `object.store(value, order)` for the first and third forms, or `func(object)` for the second and fourth. If that operation does not modify `object`, the notifying function has no effect. Otherwise, let N be the modification of `object`.
2. If `notify_all` is invoked, unblocks all executions of waiting functions invoked on `this` and `object` that blocked after observing the result of modifications that precede N in `object`'s modification order.

3. If `notify_one` is invoked, unblocks at least one execution of a waiting function invoked on `this` and `object` that blocked after observing the result of a modification that precedes N in `object`'s modification order, if any. [*Note:* Incorrect use of `notify_one` leads to deadlock. – *end note.*][*Example:*

```
struct broken_ticket_mutex {
    atomic<int> dispensing = 0, serving = 0;
    synchronic<int> sync;
    void lock() {
      sync.wait(serving, dispensing++);  // Only one thread waits for each value.
    }
    void unlock() {
      sync.notify_one(serving,
        [](auto& a){ a++; });  // error: may unblock only threads that can't continue.

      // OK: sync.notify_all(serving, [](auto& a){ a++; });
    }
  };
```

– *End Example.*]

14 *Throws:* `system_error` (19.5.6) or any exception thrown by `func`.